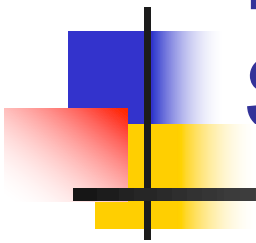


Functional Programming with Scheme

Keep reading: Scott, Chapter 11.1-11.3, 11.5-11.6, Scott, 3.6

Announcements

- Submitty autograder for HW4 working now
  - Your plan.rkt must
    - Conform to R5RS
    - Contain only function definitions
  - Submitty runs plt-r5rs v8.2:

    > (load plan.rkt) -– loads your implementation

    > (load test1.rkt) --- loads call to myinterpreter
- Exams are graded, we'll release after class
  - Original average: 72%; min: 29.5 and max: 115/115
  - I shifted curve to an average of 78%
  - Please, do write your Scheme homework :(

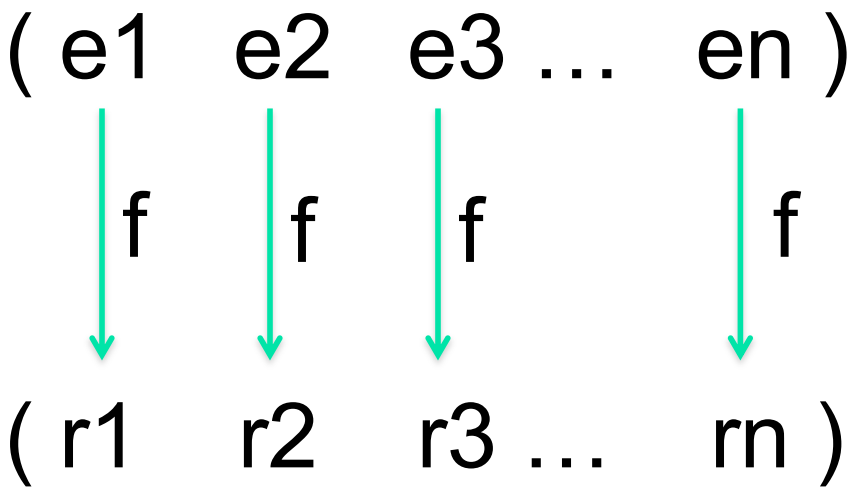
Lecture Outline

- *Scheme*
  - *Exercises with map, foldl and foldr*
  - *Tail recursion*
  - *Binding with let, let*, and letrec*
  - *Scoping in Scheme*
  - *Closures*

  - *Scoping, revisited*

map

(map f lis)



Exercises

(foldr op lis id)

(e_1 … e_{n-1} e_n) id

(e_1 … e_{n-1}) res_1

… op

(e_1) res_{n-1}

res_n

```
(define (rev2 lis)
  (foldr (lambda (el partial)
           (append partial (list el)))
         '()
         lis))
```

(list 'a) → (a)

(list 'a 'b 'c) → (a b c)

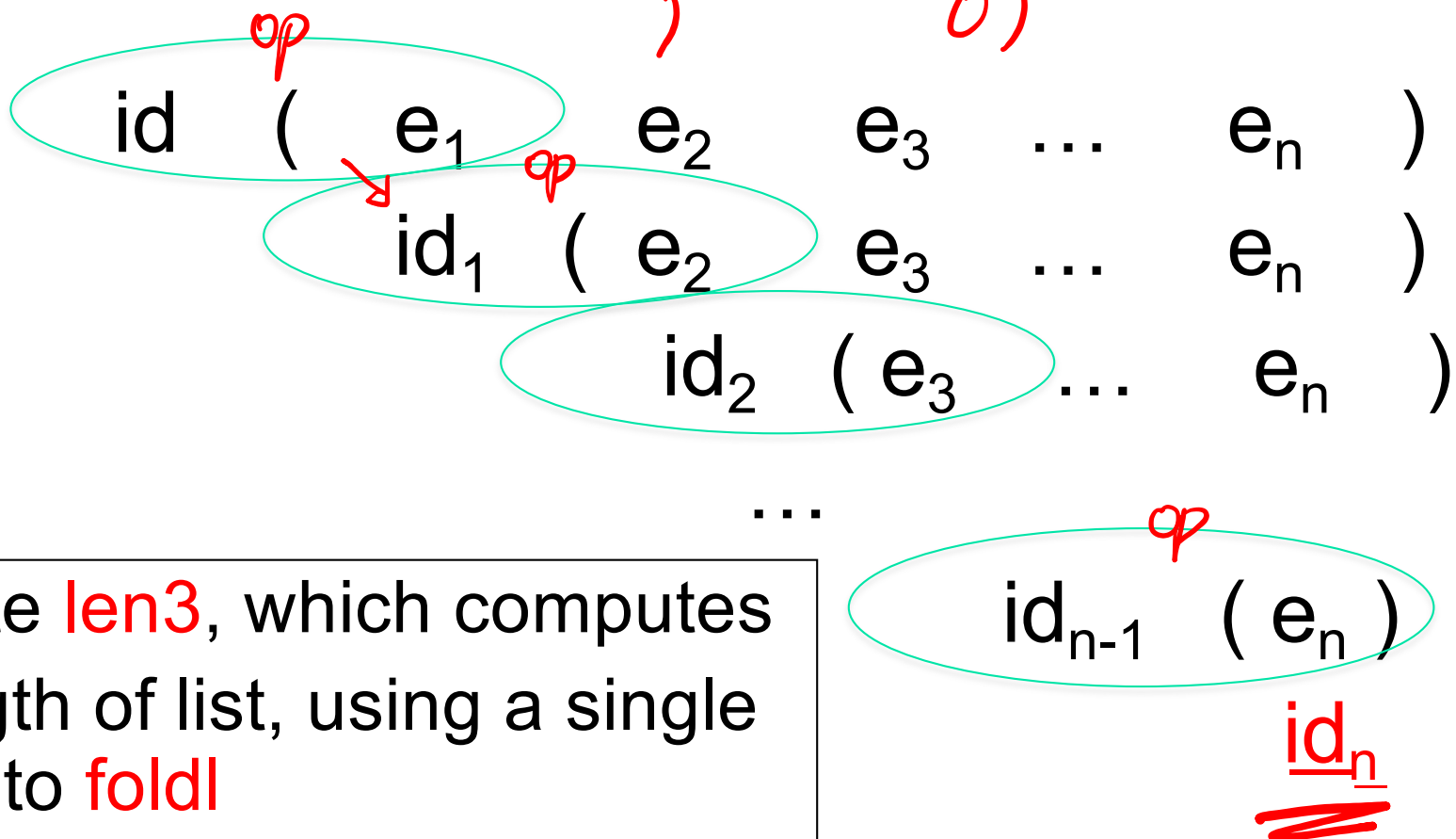
Write rev2, which reverses a list, using a single call to foldr

(define (rev2 lis) (foldr …))

Exercises

(define (len3 lis)
(foldl (lambda (partial el)
(+ 1 partial))
lis
0)
)

(foldl op lis id)



Write len3, which computes length of list, using a single call to foldl

(define (len3 lis) (foldl …))

Exercises

(flatten3 '((1) ((2) (3)))) → (1 2 3)

```
(define (foldl op lis id)
  (if (null? lis) id
      (foldl op
             (cdr lis)
             (op id (car lis)))) )
```

- Write flatten3 using map and foldl or foldr

```
(define (flatten3 lis)
   (cond ((null? lis) lis)
         ((atom? lis) (list lis))
         (else (foldl append (map flatten3 lis) '()))
   )
)
```

Implicit deep recursion!

- Write flatten4 this time using foldl but not map.

Reference implementation:

Explicit deep recursion

```
(define (flatten lis)
  (cond ((null? lis) lis)
        (atom? lis) (list lis))
        (else (append (flatten (car lis))
                      (flatten (cdr lis))))))
```

```
(define (flatten4 lis)
  (cond ((null? lis) lis)
        ((atom? lis) (list lis))
        (else (foldl (lambda (partial el)
                       (append partial (flatten4 el)))
                     lis
                     '()))))
```

foldr vs. foldl

```
(define (foldr op lis id)
  (if (null? lis) id
      (op (car lis) (foldr op (cdr lis) id)) ))
(define (foldl op lis id)
   (if (null? lis) id
       (foldl op (cdr lis) (op id (car lis))) ))
```

- Compare underlined portions of these two functions
  - foldr contains a recursive call, but it is **not** the entire return value of the function
  - foldl's recursive call is the entire return value!

Tail Recursion

- A function is tail recursive if it either returns without a recursive call or returns the result of an immediate recursive call (an informal definition)
  - E.g., foldl

- Tail recursion can be implemented efficiently
  - Result accumulates in one of the arguments, and stack frame creation can be avoided

Tail Recursion: Two Definitions of Length

```
(define (len lis)
  (if (null? lis) 0
      (+ 1 (len (cdr lis)))))

(len '(3 4 5))
```


```
(define (lenh lis total)
  (if (null? lis) total
      (lenh (cdr lis) (+ 1 total))))
(define (len lis) (lenh lis 0))

(len '(3 4 5))
```

Tail Recursion: Two Definitions of Factorial

```
(define (factorial n)
  (cond ((zero? n) 1)
        ((eq? n 1) 1)
        (else (* n (factorial (- n 1))))))
```

(n-1)!

```
(define (fact2 n acc)
  (cond ((zero? n) 1)
        ((eq? n 1) acc)
        (else (fact2 (- n 1) (* n acc)))))

(define (factorial n)
   (fact2 n 1))
```

Tail Recursion, a Bit More

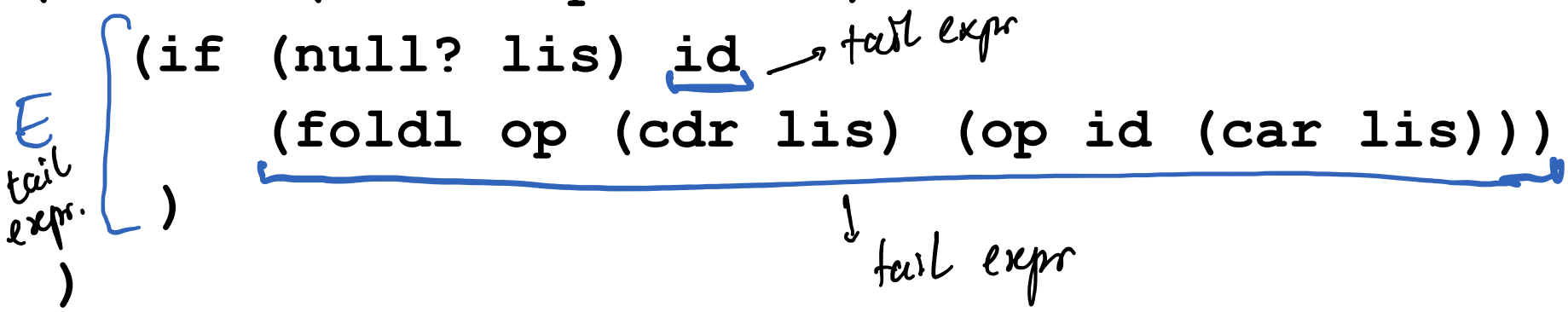
- Syntax of Core Scheme

$E \rightarrow$ `(quote` C`)`	*// constant expression*
$E \rightarrow I$	*// identifier expression*
$E \rightarrow$ `(lambda (`$I_1\ I_2\ \ldots$`)` E`)`	*// lambda expression*
$E \rightarrow$ `(if` $E_0\ E_1\ E_2$`)`	*// if expression*
$E \rightarrow$ `(`$E_0\ E\ \ldots$`)`	*// call expression*
$C \rightarrow$ `#t` | `#f` | `num` | `sym` | …	

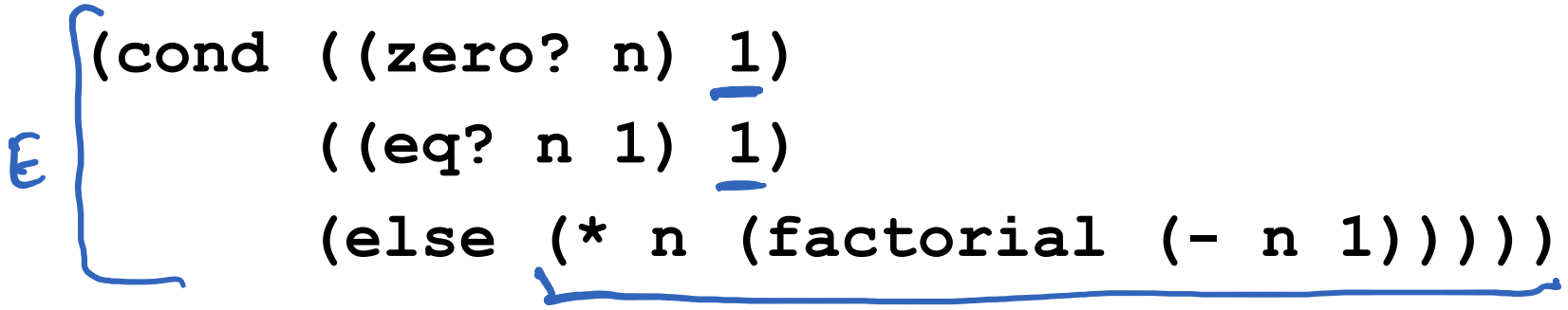
- A tail expression is an expression that occurs in tail context (in Core Scheme, can extend)
  - The body of a lambda expression is a tail expression
  - If `(if` $E_0\ E_1\ E_2$`)` is a tail expression, then E_1 and E_2 are tail expressions

Examples

```
(define (foldl op lis id)
  (if (null? lis) id
      (foldl op (cdr lis) (op id (car lis)))
  )
)
```




```
(define (factorial n)
  (cond ((zero? n) 1)
        ((eq? n 1) 1)
        (else (* n (factorial (- n 1))))))
```



Tail Recursion, a Bit More

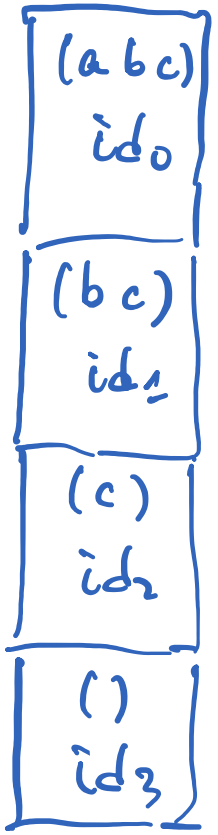
- A tail call is a tail expression that is a function call. E.g.,

```
(define (foldl op lis id)
   (if (null? lis) id
       (foldl op (cdr lis) (op id (car lis))))
  )
```

- A function is tail recursive if its tail calls are self-calls (still an informal definition)
- Tail calls give rise to efficient implementation of Continuation Passing Style (CPS)

Tail Recursion, a Bit More

foldl, normal



Creates stack frames.

tail optimization

id1 ← (op id0 'a)
lis ← '(b c)

Replaces frame of caller with frame of callee.

Supports unlimited number of active calls.

Lecture Outline

- *Scheme*
  - *Exercises with map, foldl and foldr*
  - *Tail recursion*
  - *Binding with let, let*, and letrec*
  - *Scoping in Scheme*
  - *Closures*

  - *Scoping, revisited*

Let Expressions

let, let*, letrec

Let-expr ::= (let (Binding-list) S-expr1)

Let*-expr ::= (let* (Binding-list) S-expr1)

Binding-list ::= (Var S-expr) { (Var S-expr) }

EBNF

- let and let* expressions define a binding between each Var and the S-expr value, which holds during execution of S-expr1
- let evaluates the S-exprs in Binding-list in current environment “in parallel”
- let* evaluates the S-exprs from left to right

- Associate values with variables for the local computation

Questions

`(let ((x 2)) (* x x))` **yields** 4

Binding list: `((x 2))` S-expr1: `(* x x)`

`(let ((x 2)) (let ((y 1)) (+ x y)) )` **yields ?** 3

Binding list: `((x 2))` S-expr1: `(let ((y 1)) (+ x y))`

Binding list: `((y 1))` S-expr1: `(+ x y)`

`((x 10) (y 20))`

`(let ((x 10) (y (* 2 x))) (* x y))` **yields ?**

Binding list: `((x 10) (y (* 2 x)))` S-expr1: `(* x y)`

ERROR

`(let ((x 10)) (let ((y (* 2 x))`

`(let* ((x 10) (y (* 2 x))) (* x y))` **yields ?**

Let Expressions

Letrec-expr ::= (letrec (Binding-list) S-expr1)

Binding-list ::= (Var **S-expr**) { (Var **S-expr**) }

- letrec Vars are bound to fresh locations holding undefined values; **S-exprs** are evaluated in augmented environment
- letrec allows for definition of recursive functions

```
(let ((f (lambda (x)
           (if (null? x) 0 (+ 1 (f (cdr x)))))))
  (f '(1 2 3)))
```

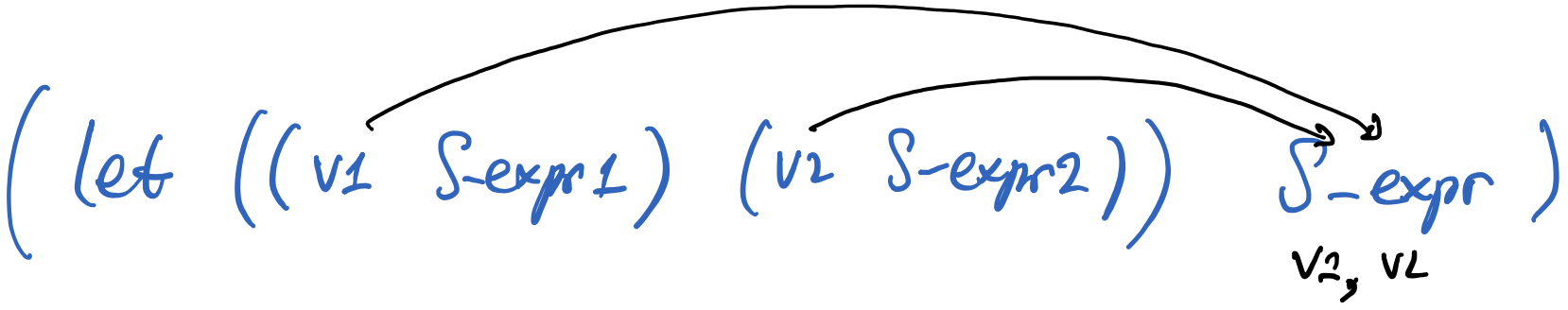
yields what?

```
(letrec
 ((even? (lambda (n) (if (zero? n) #t (odd? (- n 1)))))
  (odd?  (lambda (n) (if (zero? n) #f (even? (- n 1))))))
 (even? 88)
)
```

yields what?

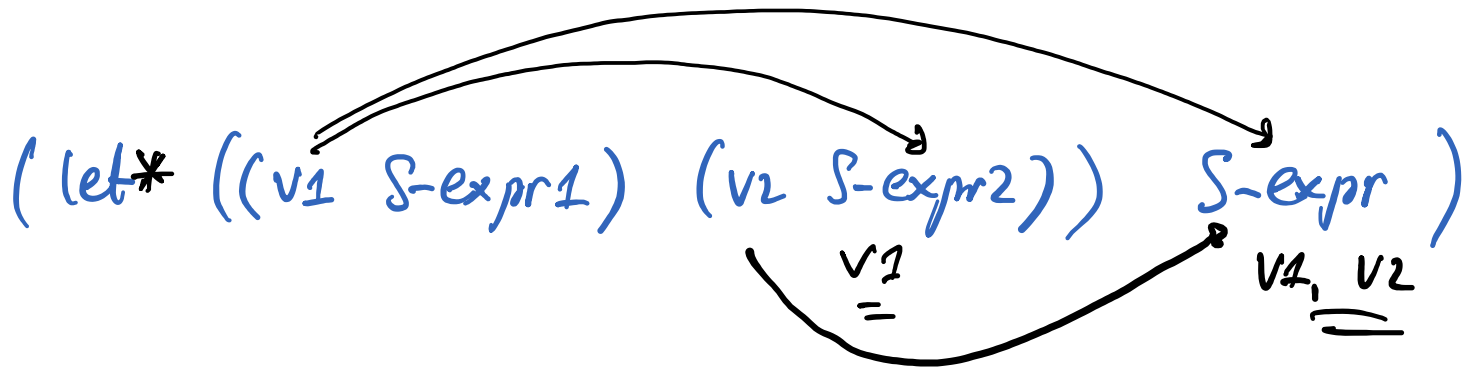
Regions (Scopes) in Scheme

- let, let* and letrec give rise to block structure
- They have the same syntax but define different regions (scopes)
- let
  - Region where binding is active: body of let



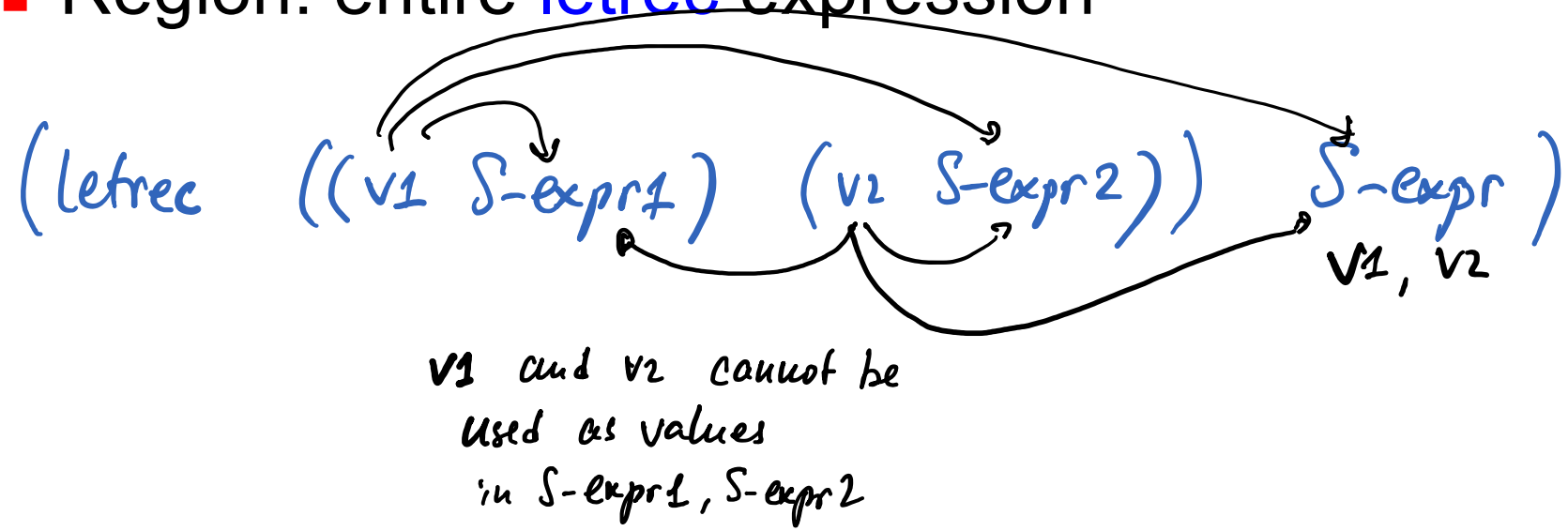
Regions (Scopes) in Scheme

- let, let* and letrec give rise to block structure
- They have the same syntax but define different regions (scopes)
- let*
  - Region: all bindings to the right plus body of let*.



Regions (Scopes) in Scheme

- let, let* and letrec give rise to block structure
- They have the same syntax but define different regions (scopes)
- letrec
  - Region: entire letrec expression



Let Introduces Nested Scopes

```
(let  ((x  10))                    ;causes x to be bound to 10
   (let   ((f (lambda (a) (+ a  x)))  ;causes f to be bound to
                                      a lambda expression
      (let ((x  2)) (f  5) ) ))
```

f is the plus-10 function

(lambda (a) (+ a x))

Assuming Scheme uses static scoping, what would this expression yield?

(f 5) → 15

plus-10 function

(f 5) → 7

plus-2 in dynamic environment

Question

```
(define (f z)
   (let* ( (x 5) (f (lambda (z) (* x z))) )
         (map f z) ) )
```

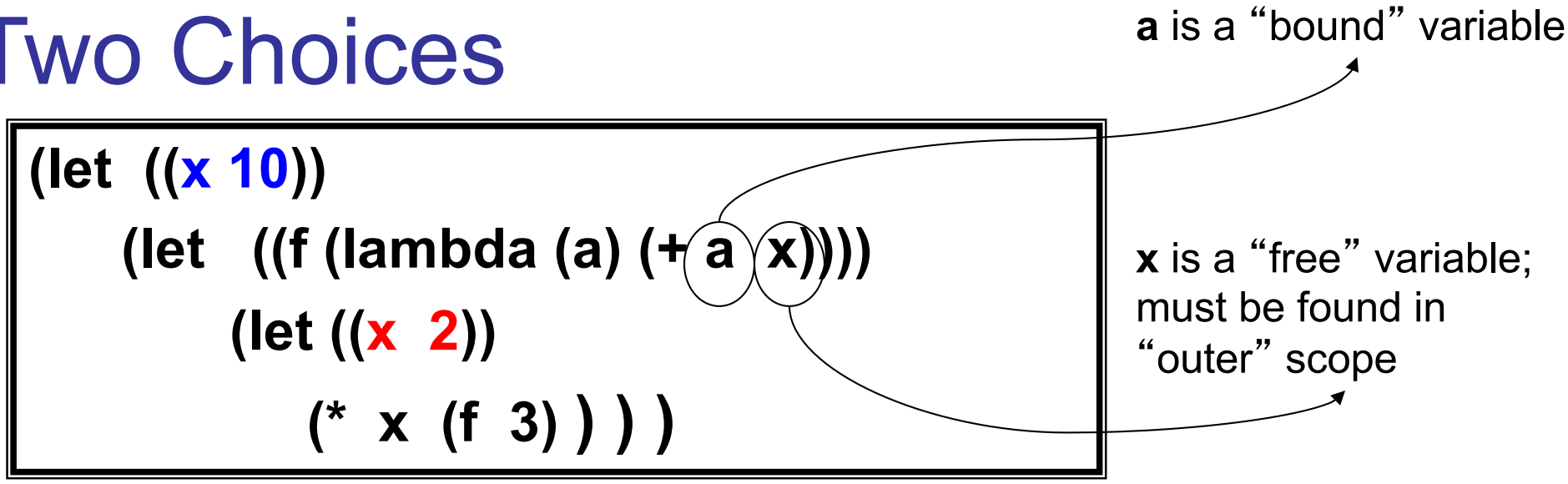
times-five

x->5

(f '(1 5 10)) → (5 25 50)

What does this function do?

Scoping in Scheme: Two Choices



With static scoping it evaluates to

```
(* x ((lambda (a)(+ a x)) 3)) -->
      (* 2 ((lambda (a)(+ a 10)) 3) ) --> ? 26
```

With dynamic scoping it evaluates to

```
(* x ((lambda (a)(+ a x)) 3))  -->
      (* 2 ((lambda (a)(+ a 2)) 3)  ) --> ? 10
```

Scheme Chose Static Scoping

```
(let  ((x 10))
     (let   ((f (lambda (a) (+ a  x))))
          (let ((x  2))
               (* x (f 3) ) ) ) )
```

f is a closure:
The function value: **(lambda (a) (+ a x))**
The environment: **{ x → 10 }**

Environment is in fact a static link.

Scheme chose static scoping:

```
(* x (lambda (a)(+ a x) 3))  -->
     (* 2 ((lambda (a)(+ a 10) 3) ) --> 26
```

Closures

- A closure is a function value plus the environment in which it is to be evaluated
  - Function value: e.g., (lambda (x) (+ x y))
  - Environment consists of bindings for variables not local to the function so the closure can eventually be evaluated: e.g., { y $\rightarrow$ 2 }
- A closure can be used as a function
  - Applied to arguments
  - Passed as an argument
  - Returned as a value

Closures

- Normally, when let expression exits, its bindings disappear
- Closure bindings (i.e., bindings part of a closure) are special
  - When let exits, bindings become inactive, but they do not disappear
  - When closure is called, bindings become active
  - Closure bindings are "immortal"

```
(let ((x 5))
     (let ((f (let ((x 10)) (lambda () x))))
          (list x (f) x (f))
     ))
```

Binding list

forces evaluation of f.

(5 10 5 10)

Lecture Outline

- *Scheme*
  - *Exercises with map, foldl and foldr*
  - *Tail recursion*
  - *Binding with let, let*, and letrec*
  - *Scoping in Scheme*
  - *Closures*

  - *Scoping, revisited*

Scoping, revisited (Scott, Ch. 3.6)

- We discussed the two choices for mapping non-local variables to locations
  - Static scoping (early binding)

  and
  - Dynamic scoping (late binding)

- Most languages choose static scoping

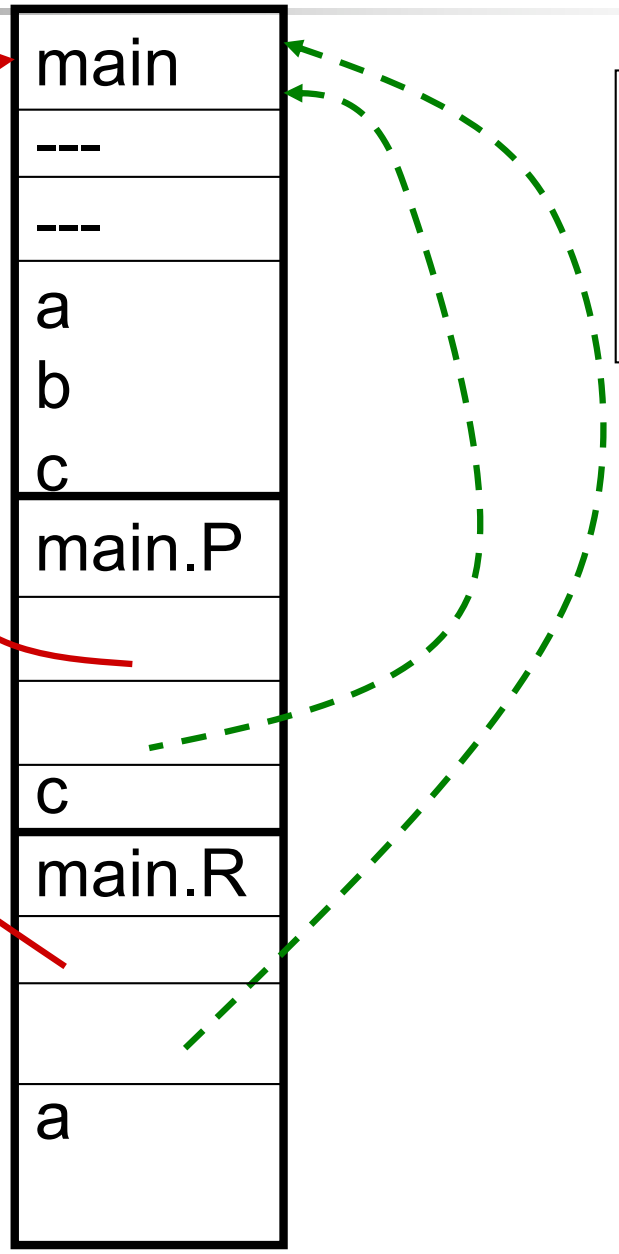
Scoping, revisited

- Earlier we assumed that functions were third-class values (i.e., functions cannot be passed as arguments or returned from other functions)

- Functions as third-class values
  - A function's static reference environment (i.e., closure bindings) is available on the stack
  - A function cannot outlive its referencing environment

Functions as Third-Class Values and Static Scoping

```
program
    a, b, c: integer;
    procedure P
            c: integer;
            procedure S
                    c, d: integer;
                    procedure R
                    …
                    end R;
                    R();
            end S;
            R();
            S();
    end P;
    procedure R
            a: integer;
            = a, b, c;
    end R;
    …; P(); …
end program
```



Static Scoping:
a bound to R.a,
b to main.b,
c to main.c

Scoping, revisited

- Functions as first-class values
  - A function value may outlive its static referencing environment
  - Therefore, we need "immortal" closure bindings
  - In languages with first-class values and static scoping, local variables must have "unlimited extent" (i.e., when stack frame is popped, local variables do not disappear)

Scoping, revisited

- In functional languages local variables typically have unlimited extent
- In imperative languages local variables typically have limited extent (i.e., when stack frame is popped, local variables disappear)
  - Imperative languages (Fortran, Pascal, C) disallow truly first-class function values
  - Increasingly, imperative languages introduce first-class functions, e.g., Java 8, C++11
  - Python has first-class functions

More on Dynamic Scoping

- Shallow binding vs. deep binding

- Dynamic scoping with shallow binding
  - Reference environment for function/routine is not created until the function is called
    - I.e., all non-local references are resolved using the most-recent-frame-on-stack rule
  - Shallow binding is usually the default in languages with dynamic scoping
  - All examples of dynamic scoping we saw so far used shallow binding

More on Dynamic Scoping

- Dynamic scoping with deep binding
  - When a function/routine is passed as an argument, the code that passes the function/routine has a particular reference environment (the current one!) in mind. It passes this reference environment along with the function value (it passes a closure)

Example

```
v : integer := 10
people : database

print_routine(p : person)
  if p.age > v
    write_person(p)

other_routine(db : database, P : procedure)
  v : integer := 5
  foreach record r in db
    P(r)


other_routine(people, print_routine) /* call in main */
```

Exercise

```
(define A
   (lambda ()
      (let* ((x 2)
             (C (lambda (P) (let ((x 4)) (P))))
             (D (lambda () x))
             (B (lambda ()
                  (let ((x 3)) (C D)))))
            (B))))
```

When we call **> (A)** in the interpreter, what gets printed? What would get printed if Scheme used dynamic scoping with shallow binding? Dynamic scoping and deep binding?

Evaluation Order

```
(define (square x) (* x x))
```

- Applicative-order (also referred to as eager) evaluation
  - Evaluates arguments before function value

```
(square (+ 3 4)) =>
(square 7) =>
(* 7 7) =>
49
```

Evaluation Order

```
(define (square x) (* x x))
```

- Normal-order (also referred to as lazy) evaluation
  - Evaluates function value before arguments

```
(square (+ 3 4)) =>
(* (+ 3 4) (+ 3 4)) =>
(* 7 (+ 3 4)) =>
(* 7 7)
49
```

- Scheme uses applicative-order evaluation

So Far

- Essential functional programming concepts
  - Reduction semantics
  - Lists and recursion
  - Higher-order functions
    - Map and fold (also known as reduce)
  - Scoping
  - Evaluation order

- Scheme

Coming Up

- Lambda calculus: theoretical foundation of functional programming

- Haskell
  - Algebraic data types and pattern matching
  - Lazy evaluation
  - Type inference
  - Monads

The End